(1)



**Interfacing the RAP System to Real-Time Control**

R. James Firby

University of Chicago
Animate Agent Project Working Note AAP-1
Version 1, July 1993



For additional copies, write to:

Department of Computer Science
University of Chicago
1100 E. 58th Street
Chicago, Illinois 60637-1504
U.S.A.





# Interfacing the RAP System to Real-Time Control

R. James Firby
University of Chicago
firby@cs.uchicago.edu
July 25, 1993

## Abstract

*This document proposes a way to integrate the RAP system with robot control systems based on concurrent interacting processes. The proposal suggests mechanisms for enabling and disabling processes without blocking, mechanisms for synchronizing with the world by blocking until appropriate signals are received from the control processes, and mechanisms for any RAP to update memory.*

## 1 Introduction

There are two aspects of the basic RAP system that make it difficult to use for controlling real robots: the assumption that control consists of sequencing atomic actions, and the assumption that only atomic actions update RAP memory.

Assuming atomic actions causes several problems. Atomic actions are not a good model for low-level robot control. It is often much easier to think of the low-level control system as a collection of concurrent, interacting processes. A robot takes a particular action by setting a subset of these processes in motion (some for sensing and some for acting) and allowing them to execute over a period of time. Within such a world view, the RAP system would be responsible for starting and stopping processes and, to a certain extent, monitoring their progress. Furthermore, real robots can often do more than one thing at a time and the RAP system should be able to pursue multiple goals concurrently as long as the hardware and control system will allow it.

Another problem with atomic actions is the implicit assumption that they have a well defined finish. Virtually all real actions fail in a variety of ways and the same action will succeed differently in different situations. The usual assumption with primitive actions is that the goal of the action is effectively independent of the current situation -- (grasp cup1) will succeed when cup1 is well in hand. However, to really grasp a cup, the RAP system may sometimes use a grasp that wraps around the cup, sometimes a grasp that pinches the rim, and sometimes a grasp that uses the handle. Each grasp will use a different shaping of the hand, a different arm motion, and different success conditions; a grasp around the cup might use palm touch sensors while a grasp of the rim may use finger tip sensors. The RAP system should be able to choose hand shapes and arm motions to create a wide variety of grasps and it should also be able to select those sensor readings that will signal success and failure.

The assumption that only primitive actions update memory is closely related to the assumption that the goals (and success and failure) of primitive actions are effectively independent of context. When a primitive action has a well defined goal and can detect its own success and failure, it can make changes in memory that reflect changes it makes in the world. It can also make inferences about the state of the world given particular primitive successes or failures. However, when the RAP system selects a set of concurrent processes for taking an action and detecting its success or failure, there is no notion of a well-defined goal. Although the RAPs that select the processes form a hierarchy that contains subgoals at the right level of abstraction for updating memory, the processes themselves don't know what is really going on. In fact, memory is hierarchical

by nature and rather than use primitive actions to try and keep the whole hierarchy consistent (as the basic RAP system does), it makes more sense for RAPs at different levels of abstraction to update memory themselves.

This document proposes a way to integrate the RAP system with robot control systems based on concurrent interacting processes. The proposal suggests mechanisms for enabling and disabling processes without blocking, mechanisms for synchronizing with the world by blocking until appropriate signals are received from the control processes, and mechanisms for any RAP to update memory.

## 2 Overview

This paper describes an interface between the RAP reactive execution system and a robot control system. There is no way to define such an interface without making some assumptions about how the control system will be configured and used. Therefore, this overview discusses a "generic" model for robot control based on the idea that different robot behaviors can be obtained by enabled different groups of control processes in different situations. Various issues involved in adapting the RAP system to manage such a control system are then presented, including the need for concurrent RAP task expansion and the need for RAP memory to be updated by RAP tasks at different levels of abstraction.

### 2.1 The Skill Model for Low-Level Control

Recently, AI researchers have proposed several different mechanisms for programming robots "reactively." These include collections of behaviors [Brooks 89], schemas [Arkin 87], routines [Gat 91], and reflexes [Payton 86]. Many details differ between these proposals, particularly in the area of philosophical commitment, but they share the common idea that the actual behavior of the robot at any given moment is the result of a set of interacting processes acting on input from the environment. Thus, the behavior of the robot (*i.e.*, its apparent immediate goal) can be changed by changing the set of active processes. This idea has been discussed by several authors and it allows some aspects of robot control to be described in terms of concurrent processes while other aspects are described in terms of discrete, symbolic steps that enable and disable those processes [Payton 90], [Gat 91], [Firby 92].

Rather than commit to a particular one of these programming models, this paper assumes a model broad enough to include them all. The general model is based on the idea of interacting processes called skills [Slack 92]. The programming model for skills, the way that skills are interconnected, and whether or not there is a fixed number of skills is not important. What matters is that skills can be treated as independent entities and enabled and disabled at will. Most models for robot programming can be phrased in terms of skills by using enable and disable to either turn on or turn off a hardwired process, subsume a process, instantiate or destroy a process, or alter the parameters of a permanent process.

Given that a desired robot action is obtained by enabling an appropriate set of skills, there is still the problem of telling when a particular goal has been attained or when a situation has arisen that prevents that goal from being attained. Sometimes the RAP system will monitor and detect these situations but often a skill will be best at detecting certain states of the world, particularly those states in which the skill is not functioning properly. Also, it will often be *necessary* to use skills to detect transient conditions reliably because skills will be able to react much faster than the RAP system. We assume that when skills detect various conditions, either good or bad, they will send asynchronous signals to the RAP

system in the form of events. Depending on the programming model used for skills, events might be generated by skill processes directly, they might correspond to particular values appearing on wires connecting skills, or they might occur when a particular skill is invoked. Events may need to be enabled separately from skills.

We also assume that some states of the skill level will be available through direct queries. It may be possible to query raw sensor values, maps or other structures built by active skills, or even the parameters and states of skills themselves.

To make use of a skill-based control system, the RAP system must be able to enable and disable skills; enable, disable, and react to asynchronous events; and translate queries from the RAP syntax into something the skill level can understand. A general interface for these processes involves two aspects: a standard set of calls that the skill level must support, and a set of translations between those skill level calls and the standard RAP syntax. The skill calls will stay the same across various control systems although the RAP translations may be different.

## 2.2 Using RAPs to Control Skills: Skill Groups

Getting a robot to do useful things requires molding skills and events into coherent groups that act to achieve a specific goal. For example, a skill for moving in a given direction while avoiding obstacles might be coupled with a skill for tracking a specific color to create a group that implements a "follow the color" behavior. Skill groups define control programs that have the property (often *emergent* ) of achieving a semantically meaningful goal. By moving from one active group of skills to another, the robot can be moved through a series of low-level goals. In effect, skill groups can be used to implement discrete primitive actions.

RAPs define a hierarchy of methods for carrying out discrete (although possibly concurrent) tasks in the world. The notion of discrete tasks and subtasks matches the way task plans are usually conceived, gives the RAP methods a clear semantics, and is crucial when RAPs and RAP tasks are used as planning operators. Skill groups are a way to map discrete RAP goals onto a continuous control system. When the RAP system refines goals into a sequence of skill groups, it is dividing behavior into segments that can each be implemented with a single configuration of the control system.

There are several aspects to defining a skill group that it implements a reliable subgoal for the RAP system. First, a group must implement a well-defined goal so that it can be used meaningfully in plans. This is not an onerous requirement because the RAP system can easily use groups that achieve a goal only under some circumstances. Second, when a group's goal has been achieved, it must signal success and transition to a mode of behavior that will maintain the goal until the RAP system has time to activate the next skill group. This aspect of skill group design is the most difficult to get right. Finally, a skill group must signal when it is stuck, diverging from its designated goal, or failing in some potentially dangerous way. While the RAP system is capable of monitoring the progress of skills (by monitoring sensor information) it will typically be operating much more slowly than the skill level and can supply only loose supervision. It is important that skill groups detect success, failure and inefficiency themselves. Only when skill groups are *closed* in this way under both success and failure can the RAP system use them with confidence. Unfortunately, defining a useful set of closed skill groups is currently a black art.

Given a closed skill group, the RAP system can activate the constituent skills and events, and then wait for either a success signal or a failure signal. If the group is truly closed a signal is guaranteed. This simple notion provides the model for the basic programming

construct that connects the RAP and skill systems. A form is provided for defining primitive RAP actions that activate a group of skills and then wait for an event to be signaled. When a signal is received the primitive action is assumed to be complete, a result is returned based on the signal received, and RAP execution resumes. Most activity can be expressed with this construct. However, under some circumstances, it makes sense to activate skills individually and watch for various events and sensor readings at the RAP level. This way of organizing and representing behavior is also supported by the same constructs.

## 2.3 RAPs and Concurrency

The RAP system must support concurrent processing of the RAP task agenda to enable RAPs to activate individual skills and events and then watch for changes in the world. The initial RAP system assumed that all activity consisted of individual, sequential primitive actions and that RAP processing could stop during the execution of a primitive without harm. Within the skill model, RAPs can activate and deactivate individual skills this way, but after an entire skill group is activated it must run for some extended period of time to accomplish a task. During this time, the RAP system can neither stop execution entirely, nor forge blindly ahead expanding tasks down to skill activations. Once a required set of skills are active, the RAP system must wait for them to succeed or fail before going on to the next subtask in that method but it should continue to execute tasks in other task families.

The critical step in concurrent RAP task processing is not getting RAPs to run concurrently, but stopping expansion in the right places to synchronize processing with the actual progress of the task in the real world. This synchronization is achieved by extending the RAP system to wait for asynchronous events by including statements in RAP task-nets that cause processing of the method to pause until a specific event occurs in the world. Presumably such events will be generated by skills that were enabled previously. While one RAP method is blocked waiting for an event, other RAP tasks that can run concurrently will continue to run.

Another issue for a concurrent RAP system is dealing with conflicts that arise when two RAPs require access to the same resource (*i.e.*, a gripper, an arm, power, computation, the camera direction, or some object out in the world). Typically these conflicts involve different tasks wanting incompatible skills to be active together. We define two rules for dealing with RAPs that request incompatible primitives: (1) RAPs with higher priority override RAPs of lower priority and (2) a RAP enabling a skill that conflicts with an enabled skill at the same or higher priority fails. This default behavior can be altered somewhat but essentially all conflicts are detected and dealt with at the primitive skill level.

Conflicts that involve the use of external resources must be taken care of by the RAP code or a high-level planner. For example, a particular sort of insidious conflict comes from expanding RAPs from multiple families in the wrong context simply because the tasks were available for expansion while the correct next step to execute was blocked waiting for an event to occur. It is imperative that top-level RAP tasks that are not ordered with respect to one another truly be independent. Such independence is almost never actually true; leaving two tasks unordered in the RAP system usually does not mean they can be executed in parallel, it means they can be executed in any order. When RAP expansion occurs during primitive action execution, it is critical to distinguish truly concurrent tasks from tasks that can merely be executed in any order. The RAP task-net syntax must be extended to allow the specification of true concurrency and the default semantics of unordered tasks is changed to mean they can be executed in any sequence.

### 2.4 RAPs and Memory

Another important issue in porting the RAP system to manage a skill-based control system is updating memory in response to RAP activity. In the initial implementation of the RAP system, all memory changes were made by the primitive action and sensor handlers; RAPs and the RAP interpreter never changed memory themselves. This approach to memory maintenance is clearly too simplistic. When primitive actions exist and have a well defined semantics in the world, it makes some sense to put memory updating down at the level of primitive action execution because the interpretation of action results is independent (to a large extent) of the task hierarchy in which the action is executed. However, when primitive actions are synthetic constructions, their semantics is known only to the task that generated them. Thus, the RAP system must allow RAP tasks to update memory in response to events and sensor data generated by the skill system. The dynamic, RAP expansion hierarchy that enabled the skills involved holds the context required to properly interpret signals from the skills.

The ideal model of memory update is for the RAP system to simply record its actions (*i.e.*, the activation, expansion, and results of RAP tasks) while a separate "understanding" system considers the RAP system's actions along with all other sources of knowledge available to make the best inferences possible about what is happening out in the world. Constructing such a system is a big job and will take some time [Martin and Firby 91]. In the meantime, a simpler but conceptually compatible mechanism for memory update will have to do.

The proposed memory update mechanism is based on RAP activation and completion. Each time that a RAP task is activated (*i.e.*, is selected for execution the first time) or completed (*i.e.*, removed from the task agenda), a memory update rule specific to that task type is run. The rule can examine the current contents of memory and the result of the execution of the RAP and then alter memory any way that it chooses. This mechanism allows the results of task execution to be interpreted in the correct context and at the correct level of abstraction. Memory rules can also be defined to be run when a particular event occurs during the execution of a task.

Running memory update rules on RAP task activation, completion, and event detection allows arbitrary inferences to be made in response to RAP task execution. Unfortunately, this mechanism does not support a more global view that could interpret events that take place outside of the current execution context. Truly unexpected events will always go unnoticed unless they directly effect the tasks currently underway. A wider view requires a separate system dedicated to trying to understand what's happening in the world.

## 3 The Standard Skill Level Interface

The standard skill level interface defines the functions that must be implemented by the skill level. The skill level (*i.e.*, the control system) can implement these functions any way that it chooses as long as each function adheres to the behavior described. The skill level is free to do nothing if a function call doesn't make sense given the particular skill programming model. Typically RAPs will be different when the skill system implements these functions in different ways.

### 3.1 Enabling and Disabling Skills

The whole skill model is based on the idea of enabling and disabling skills to carry out various tasks. Thus, the skill level must support the following two calls:

(enable-primitive-skill skill-definition enabled?) => boolean
(disable-primitive-skill skill-name)

A skill-definition must be a list with the first element naming the skill to be enabled and the rest of the list giving any parameters the skill requires. The enabled? flag is a boolean value that will be T if it is okay for the named skill to already be enabled and nil otherwise.

If enable-primitive-skill is called and the named skill is already enabled and it isn't okay (*i.e.*, enabled? is nil) an unexpected skill conflict has occurred. When this happens, the skill level should ignore the current call and return NIL. Otherwise enable-primitive-skill should return T. It is never a problem for a skill to be disabled even if it has not been enabled.

(fetch-skill-resources skill-name) => list-of-symbols

When primitive actions are defined, the RAP system must find out which skills conflict. It does this by assuming that each skill makes use of some set of resources. When two skills use the same resource they are assumed to conflict. The RAP system asks for the resources each skill uses by calling fetch-skill-resources. It assumes that a list of resource names (*i.e.*, symbols) will be returned.[1] Note that the symbols have no meaning to the RAP system; skills can use any symbols that are convenient.

## 3.2 Enabling Skill Level Events

The skill model assumes that events must be enabled before they will generate asynchronous results. All events can be configured in either single shot or continuous mode. Single shot events effectively disable themselves after being triggered once.

(enable-primitive-event event-id :single/:continuous event-definition event-values)
(disable-primitive-event event-id)

Events are enabled by calling enable-primitive-event with a unique event-id, a keyword to describe whether :single or :continuous mode is desired, the definition of the event to be enabled and a list of values that should be captured at the time of the event. The format for event definitions and event values is up to the skill level.[2] A corresponding disable-primitive-event call will be used to disable continuous events when required.

(fetch-primitive-events) => list-of-event-id/value-exps

Even though the skill level will generate asynchronous events, the RAP system can only dispatch on these events at particular points in its execution cycle. Thus, the skill level is required to save up events and return them only when the RAP system asks using fetch-primitive-events. This function must return a list of events. Each event is a list with the first element being the event-id of the event triggered (from enable-primitive-event), and the rest being a list of the requested event values. Event values are expected to be captured at the time the event is triggered.

---

[1] I can imagine a whole slough of different types of resources and a huge variety of rules for detecting conflicts. I intend to simply ignore this problem for the time being. Conflict detection through this mechanism is really the last line of defence. Conflicting primitive actions should be specified specifically using (suspend ...) and (conflicts ...) clauses.

[2] However, the translation layer between the RAP system and the skill level requires that skill definitions, event definitions and skill values all be parsable as LISP atoms or lists. This restriction is necessary so that RAP variables in these definitions can be replaced by vtheir alues before the skill level functions are called.

### 3.3 Querying the Skill Level for Information

The skill level can be asked for information through the call :

(query-primitive-value query-definition query-values) => list-of-out-var-values

The first argument to this function is a definition for the query itself and the rest of the arguments are either values being passed in or values to obtain from the skill level (like event-values). The *query-definition* and *query-values* do not mean anything to the RAP level, they are used only by the skill level. Typically, the query-definition will be a symbol and the query values will either be values supplied to that query or values that the query is supposed to return. The query is expected to return a list of values with the same number of entries as the list of query-values passed in.[3]

## 4 Connecting RAPs to the Skill Level

While the skill level function calls defined above supply the interface to the skill system, there are many additional issues involved in mapping RAP level memory structures and variables onto those calls. The macros described in this section allow the user to define this mapping in a reasonably general fashion. Basically these forms specify syntactic transformations between RAPs and skills. The exception is define-primitive-action which incorporates many RAP forms to control execution.

Effectively, these functions replace the primitive action handlers used to carry out actions in the basic RAP system. Once an action is defined using define-primitive-action, it can appear in any RAP method just like another RAP task. Similarly, once an event is defined using define-primitive-event, it can appear in any RAP monitor or wait-for clause and once a query is defined using define-primitive-query, it can appear in any RAP query. The difference is that these actions and queries don't generate other RAP tasks or memory queries, they act on the skill level instead.

### 4.1 Define-Primitive-Action

The only way that the RAP system can enable skills (and hence initiate action) is by executing a task that names a primitive action. All skill enabling and disabling can be done one at a time with the standard RAP task-net annotations (described below) used to coordinate and synchronize with the external world. However, as a useful shorthand, define-primitive-action allows skills to be enabled and disabled in sets. This approach is only marginally more efficient but it is often much more convenient.

```
(define-primitive-action (action-name in-vars => out-vars)
    (preconditions query)
    (timeout delta-time optional-result)
    (compatible list-of-action-names)
    (suspend list-of-actions-to-suspend)
    (conflicts list-of-action-names)
    (enable list-of-skill-activations)
    (disable list-of-skill-names)
    (wait-for event succeed/fail optional-result) ...)
```

The preconditions and timeout clauses are the same for a primitive action as they are for a normal RAP. If the preconditions are not true,or if the timeout expires before an event

---

[3] Query values (and event values) are derived from RAP queries. Define-primitive-query describes the mapping between RAP variables and values and the values passed to and returned by query-primitive-value.

occurs, the action will fail. When the primitive action is executed, the RAP interpreter will replace any bound variables in the skill and event lists with their corresponding bindings and call the appropriate number of enable-primitive-skill. No unbound variables are allowed When the primitive action completes (the RAP system will figure this out) the skills mentioned in the disable clause will be stopped using disable-primitive-skill. When a wait-for clause is present, the event mentioned in the clause is enabled (in :single shot mode) and the primitive action does not complete until that event is received. If multiple wait-for clauses are present, the primitive will complete when any one of the events occurs. If no wait-for clauses are present, the primitive completes immediately.

All enabled events are disabled when the action completes but enabled skills are not disabled unless explicitly mentioned in the disable clause. Since it is common to disable all enabled skills when an action completes, the special symbol ':above can be used in the disable list to indicate all skills enabled "above" are to be disabled.

Conflicts in skill activations can arise when RAPs enablesprimitive actions that use the same skills. When a conflict occurs, the RAP primitive with the higher RAP priority overrides the other. A higher priority primitive RAP started first continues to run and the RAP primitive started second fails. A higher priority RAP started second causes the earlier RAP primitive to fail (disabling it). RAPs that fail due to skill conflicts return the result 'conflict. This default conflict resolution behavior can be altered using the (suspend ...) and (compatible ...) primitive action definitions. A primitive naming other primitives in a suspend clause will override those primitives (causing them to fail) regardless of their current priority. A primitive naming other primitives in a compatible clause will simply coexist with the other primitives, neither causing the other to fail. The parameter values of later compatible primitives override the parameters of earlier ones. The (conflicts ...) clause can be used to specify conflicting actions at the action level. The order of precedence when primitives are named in several clauses is: suspend, compatible, and default behavior.

Examples:

```
(define-primitive-action (wander ?timeout)
    (timeout ?timeout)
    (enable (:wander))
    (disable :above)
    (compatible runaway))

(define-primitive-action (localize-tag ?tagnumber ?confidence ?timeout)
    (timeout ?timeout tag-not-found)
    (preconditions (active-skill :track-scanner-targets))
    (enable (:localize-tag-snat  ?tagnumber)
            (:track-sonar-obstacles)
            (:compute-objconfig))
    (disable :above)
    (compatible runaway)
    (wait-for (tag-location-found ?tagnumber ?confidence) :succeed tag-found))
```

Note that the timeout clause is almost always used to guarantee, in the crudest way, that the action being defined generates a closed skill group.

## 4.2 Define-Primitive-Event

This form defines the mapping between RAP forms and skill level events. Basically, the form gives the event-description and event-values that should be used in the call to enable-primitive-event when a wait-for clause is encountered that specifies the event.

```
(define-primitive-event (event-name vars)
   (event-definition sexp)
   (event-values sexp-for-each-var)) ; :bound means var must be bound
```

The sexp in the event-definition is entirely defined by the low-level skill system. The RAP system will replace any unbound variables in the sexp with their bindings before passing it to the enable-primitive-event function. The RAP system will also replace any bound variables in the event-value sexps with their bindings and pass the resulting list to enable-primitive-event. The skill system is responsible for returning the correct values for each var sexp passed in as part of the generated event. These values must be in the same order in the event as they are in the sexp list. A ':bound sexp signals that the corresponding var in the event must be bound before the event is enabled (the RAP system will check this before calling enable-primitive-event).

Examples:

```
(define-primitive-event (power-sensed ?state)
   (event-definition
      (primitive-value :basepower :valid))
   (event-values
      (primitive-value :basepower :state)))

(define-primitive-event (runningaway)
   (event-definition
      (becomes-true (primitive-value :runaway :valid))))
```

Note that when an event is enabled, it is given a unique-id. That id is what gets passed to the low-level system, not the RAP name for the event. The RAP system will keep track of enabled events and skills.

## 4.3 Define-Primitive-Query

Primitive queries work almost exactly like primitive events except that the RAP system calls query-primitive-value and expects the values to be returned immediately in a list rather than queued.

```
(define-primitive-query (query-name vars)
   (query-definition sexp)
   (query-values sexp-for-each-var)) ; :bound means must be bound
```

There must be a query value sexp for each var in the RAP language query. The RAP system will replace variables with bindings and call query-primitive-value with the resulting list (':bound again means the variable in the RAP query must be bound). The returned list of values is matched against RAP unbourd variables and the RAP variables are bound.

Examples:

```
(define-primitive-query (configuration ?xpos ?ypos ?angle ?headangle)
   (query-definition configuration)
   (query-values (:ubob-config :ypos)
                 (:ubob-config :ypos)
                 (:ubob-config :angle)
                 (:ubob-config :headangle)))

(define-primitive-query (tag-reading ?tag ?angle)
   (query-definition barcode-value)
   (query-values :bound
                 (:scanner ?tag)))
```

## 5 Updating RAP Memory

A new feature of the RAP system is a way to update memory as a result of task starts, finishes, and the occurrence of events within tasks. The updates to perform at each of these times are specified in memory rules. Memory update rules roughly correspond to the primitive action and sensor handlers used by the basic RAP system. However, update rules can be defined for any RAP, not just the primitives defined with define-primitive-action.

An important feature of the old primitive action handler model was that the handler could update memory differently when the primitive action generated different results. The idea of actions producing different results (particularly on failure) is critical to keeping a useful, consistent memory. However, skills don't generate results, they generate signals that mean different things in different contexts. Thus, RAPs must map signals (and other methods for task completion) into "results" that can be used in updating memory.

A task result consists of two pieces of information: the task state, and the task result. The mapping of signals to states and results is taken care of by the wait-for clause in primitive action definitions and task-nets. Each wait-for clause specifies the task state and result to use if the signal beging waited on occurs. The timeout clause also allows a result to be specified since it is effectively a wait-for failure clause in disguise. RAP tasks can complete for various reasons other than receiving signals and there are other task states to capture these reasons. A task may evaporate because it is part of a task-net that completes for some other reason. A task may fail because it attempts to execute a primitive action that conflicts with another, there is no applicable method, it is in a futile loop, or its preconditions aren't met. Possible task state are:

| | |
|---|---|
| success | - Any successful task completion |
| failure | - Any unsuccessful task completion |
| evaporate | - A task is terminated without completing |
| conflict | - Primitive actions conflict |
| futile-loop | - A futile loop is detected |
| no-method | - No applicable method exists for a task |
| constraints | - A precondition or constraint is violated |
| timeout | - A timeout form has expired |

The rules for propogating task results upward in the RAP hierarchy are somewhat arcane.[4] If a primitive action terminates due to a wait-for or timeout clause, the result of that clause is the result of the action. If a non-primitive RAP completes, it will return either the result of the last task-net executed or the result of a timeout clause if that caused the failure. The result of the last task-net executed will be either the result from any wait-for clause in the net that caught a success or failure signal or simply nil if there are no wait-for clauses.

Update rules are defined using the form:

```
(define-memory-rule (rap-name vars) :start/:finish/:event
   sexps)
```

Memory rules are called just before the start of the task, just after the finish of the task, or when an enabled event occurs during the execution of the task. The sexps in each memory rule can be arbitrary LISP code (not much of a theory at this point) but most of the time they need to include only the match-result and rule forms. When a task finishes

---

[4] Which almost certainly means they aren't correct.

or an event occurs, the state and result of the task or the event are available and can be matched with the form:

```
(match-result
   (result sexps)
   (result sexps) ...)
```

A match-result form looks like a cond clause in LISP but it unifies the result or state of the task (or the event that occurred) with the form at the beginning of each clause, binding variables. The arbitrary sexps within each matching clause are then executed. If more than one clause matches, all are executed. Task results and states are discussed above.

A memory rule can also include the rule form:

```
(rule (<query> effect-forms) (<query> effect-forms) ...)
```

This form also looks like a LISP cond clause but for each form it does the memory query, binding variables, and if the query succeeds, executes the effect forms. Like cond, only the effect forms of the first successful query are executed. Possible effects are those defined for tasks in RAP system task-nets augmented with the new mem-add and mem-del effects giving the list:

```
(mem-add clause)
(mem-del clause)
(counter-add counter)
(counter-sub counter)
(set-add set)
(set-sub set)
```

In fact, mem-add and mem-del are added to task-net effects as well, allowing task-nets to make simple modifications to memory directly.[5]

A memory rule can also include the var-let form:

```
(var-let ( (lisp-variable rap-variable) (lisp-var rap-var) ...) lisp-body)
```

This form is similar to LISP let except that it allows rap variables to appear as evaluated forms.

Examples:

```
(defun handle-moving-robot ()
   (rule T
      (mem-del (facingportal))
      (mem-del (throughportal))
      (mem-del (atportal))
      (mem-del (attag))))

(define-memory-rule (approach-scanner-tag ?tag ?dist ?time) :start
   (handle-moving-robot))

(define-memory-rule (approach-scanner-tag ?tag ?dist ?time) :finish
   (match-result
      (succeed
         (rule T (mem-add (attag ?tag))))))
```

---

[5] I'm not convinced that allowing task-nets to modify memory is a good idea. However, it seems harmless in the hands of experienced professionals. Just don't try it at home.

```
(define-memory-rule (exit-room ?maybeadoor) :finish
   (match-result (succeed
      (rule (and (throughportal ?portal)
                 (current-location ?room)
                 (connect ?room ?someplace ?portal))
            (mem-add (current-location ?someplace))))))
```

## 5.1 Extensions to the Memory System

The RAP memory system is also extended with the following concepts:

(declare-memory-fluent name)

A fluent defines a memory proposition (not an item property) that can only be single valued. Asserting a fluent proposition deletes previous instances of the same fluent and adds the new assertion.

(declare-memory-resource resource-definition)

*e.g.* (declare-memory-resource name (counter initial-value))
*e.g.* (declare-memory-resource name (set initial-value))

This form allows the user to define a global resource. The resource definition is one of those defined for RAP resource clauses. The difference is that the name is not a variable but a global symbolic name for the resource. A global resource can be accessed in a query using the form:

(resource-type name) *e.g.*, (counter name) *or* (set name)

The memory system is also augmented with functions to check whether a given low-level skill has been enabled:[6]

(skill-enabled skill)

# 6 Modifications to the Basic RAP System

So far this proposal has not suggested any significant changes to the RAP syntax or semantics aside from the addition of a few minor constructs. In fact, the basic RAP system works quite well and need only be changed to allow blocking to synchronize with the world. However a number of other small changes are also proposed to make the system a little more flexible and easy to use. In particular, the following top-level RAP definition forms are added:

(timeout delta-time optional-result)

The timeout clause causes the RAP to fail if it does not finish within delta-time. If the RAP fails because of the timeout, it will fail with the optional-result and the state 'timeout.

(futility-threshold count)

---

[6] I'm not convinved these forms are reasonable either. The notion of a skill is only semi-defined. Perhaps it is harmless.

This clause sets the number of times the task may try each of its methods without success before a futile loop is declared. This clause overrides the default value set within the RAP interpreter.[7]

(monitor event/memory-query [:continuous])

The monitor clause is changed to take an event or a set of events combined with AND and OR. The events may be primitive events or memory queries. A monitor clause will enable its event (or events) and disable them when a single event occurs (*i.e.*. it enables primitive events with the :single event keyword). If the :continuous keyword is added to the monitor clause, it will not disable its events each time it is activated. Instead, it will leave the event activated (*i.e.*. it enables primitive events with the :continuous keyword) and will turn futile loop detection off for the task (*i.e.*. it effectively sets the futility threshold to infinite).

In addition to these new RAP clauses, the reasons section of task-net task definitions is extended to include the mem-add and mem-del effects as well as the synchronization wait-for clause and the concurrent-with clause:

(wait-for event/memory-query [:succeed/:fail] [result])
(concurrent-with task-id)

This clause causes the task to succeed (or fail) should the designated event occur before the task ends in the normal way. The state of the task is either 'succeed or 'fail, and the result is the optional form which can refer to bound variables. The result will be used in memory-rules for the enclosing RAP. This wait-for in a task-net is essentially identical to the wait-for used in define-primitive-action.

The concurrent-with clause states that this task is supposed to run concurrently with the specified other task in the task-net. Unordered tasks that are not designated as concurrent will be executed in an unspecified sequence.

Task-nets are also extended to allow for nil task-nets. A nil task-net always succeeds.

(task-net nil)

In addition to the above changes, the RAP syntax now requires that the index clause come first. With this requirement, the form (index (rap-index ...)) can be shortened to (rap-index ...), allowing the following:

(define-rap (rap-name args)
...)

## 8 Reservations about this Proposal

• This model for updating memory is incorrect. RAPs should not update memory themselves, some other process should do the job and it should be informed of when RAPs start and finish.

• Primitive actions should allow one group of skills to transition automatically into another group on certain events. That way, a robot could transition into some sort of safe holding action while waiting for the RAP system choose the next primitive.

---

[7] This notion of allowing different RAPs to have different futile loop thresholds is still a hack. It is standing in for the more sophisticated understanding of what is going on that a good planner/understander would supply.

• The whole task-net notion may have to be reevaluated somewhat to account for interesting new work by Tim Converse. In Converse's planner, tasks are not sequenced explicitly. Rather, tasks have triggering states that cause them to run. RAPs and task-nets should be altered to support this behavior.

• It seems like a toss up whether there should be memory rules for events that occur without having an existing RAP task as their context. Certain events occur accidentally and should trigger memory updates even if no RAP is currently expecting them. Currently one would need to set up a task monitoring the event and a memory-rule for the task. Clearly that is a hack. However, this problem arises out of the lack of an understanding process, not as a real shortcoming of the RAP system. A cleaner hack might be nice though.

## References


[Brooks 89] - *"The Behavior Language; User's Guide"* , Rodney Brooks, Seymour Implementation Note #2, MIT AI Lab., October 1989.

[Arkin 87] - *"Motor Schema Based Navigation for a Mobile Robot: An Approach to Programming by Behavior"*, Ronald C. Arkin, IEEE Conference on Robotics and Automation, March 1987.

[Gat 91] - *"Autonomous Goal-Directed Reactive Control of Autonomous Mobile Robots"*, Erann Gat, Ph.D. Thesis, Computer Science and Applications, Virginia Polytechnic Institute, 1991.

[Payton 86] - *"An Architecture for Reflexive Autonomous Vehicle Control"*, David Payton, IEEE Conference on Robotics and Automation, 1986.

[Payton 90] - *"Exploiting Plans as Resources for Action"*, David Payton, DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control, November 1990.

[Firby 92] - *"Building Symbolic Primitives with Continuious Control Routines"*, R. James Firby, First International Conference on AI Planning Systems, June 1992.

[Slack 92] - *"Sequencing Formally Defined Reactions for Robotic Activity: Integratings RAPS and GAPPS"*, Marc G. Slack, SPIE Conference on Sensor Fusion V: Simple Reasoning for Complex Action, November 1992.

[Martin and Firby 91] - *"Generating Natural Language Expectations from a Reactive Execution System"*, Charles Martin and R. James Firby, Thirteenth Annual Conference of the Cognitive Science Society, August 1991.